EV355228731

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

MAPPING BETWEEN STRUCTURED DATA AND A VISUAL SURFACE

Inventors:

Prakash Sikchi

Evgeny N. Veselov

and

Stephen J. Mooney

ATTORNEY'S DOCKET NO. MS1-1558US

TECHNICAL FIELD

This invention relates to the rendering of documents containing structured data, and, in a more particular implementation, to the rendering of structured data in a manner that provides mapping between the structured data and a visual presentation of the structured data.

BACKGROUND

Fig. 1 shows a typical apparatus 100 for processing documents containing structured data expressed using the Extensible Markup Language (XML). The apparatus 100 includes an Extensible Stylesheet Language (XSL) processor 102 that translates an XML document 104 into a transformed document 106. The transformed document 106 can comprise another XML document, or a document expressed in a presentation-oriented markup language, such as Hypertext Markup Language (HTML). XML provides tags that represent the subject matter contained in a document. In contrast, presentation-oriented languages, such as Hypertext Markup Language (HTML), provide tags that convey the visual appearance of a document. Accordingly, these technologies complement each other; XML allows information to be efficiently transferred and processed, while HTML allows information to be presented for display.

XSLT itself uses the syntax of XML. The XSLT processor 102 performs its translation function by making reference to one or more style sheets 108. The style sheets 108 contain a collection of rules for transforming elements in the input XML document 104 into the transformed document 106. To perform this function, XSLT relies heavily on XPath functionality. XPath is a general-purpose query notation for addressing and filtering the elements and text of XML documents. XPath expressions can address parts of an XML document, and can manipulate strings, numbers, and

Booleans, etc. In the context of the XSLT processor 102, XPath expressions can be used to find a portion of the XML document 104 that matches a prescribed match pattern, and then perform some translation operation on that portion using a rule provided in the style sheets 108. XML, XSL, and XPath are described at length in their governing specifications provided by the World Wide Web Consortium (W3C).

The translation function provided by the XSLT processor 102 is strictly one-way. In other words, the XSLT processor 102 efficiently translates the structured data in the XML document 104 into the transformed document 106. But conventional XSLT does not also provide a mechanism for translating the transformed document 106 back into the XML document 104 from which it is derived. More specifically, it can generally be said that a collection of elements in the transformed document 106 are derived from or based on one or more elements in the XML document 104; however, there is generally no way of discovering this nexus once the XML document 104 has been translated into the transformed document 106. This situation is akin to the scenario in which a file containing source code expressed in human readable form is transformed into executable code using a compiler. It may be impossible to determine the source code simply by examining the resultant executable code. The one-way nature of the translation of the XML document 104 into the transformed document 106 is represented in Fig. 1 by the arrow 110.

The one-way nature of the translation 110 performed by the XSLT processor 102 introduces difficulties in applications that demand two-way interaction between the XML document 104 and the transformed document 106. For instance, an HTML document may include a collection of fields for receiving data entered by an editing user. If this HTML document is based on an underlying XML document, it would be desirable to provide a mechanism for routing the user's input back to the source XML document. As

explained above, bare XSLT does not provide the intelligence to provide this functionality.

As such, there is an exemplary need in the art for a data processing application that provides mapping between structured data and a visual surface used to display the structured data.

SUMMARY

According to one exemplary implementation, a method is described for mapping between parts of an input document and associated parts of an output document. The input document pertains to a first kind of document (such as XML), and the output document pertains to a second kind of document (such as HTML). The method includes: (a) providing a translation file (such as XSLT) that converts documents of the first kind to documents of the second kind; (b) in a first phase, modifying the translation file to include mapping functionality that can provide information regarding relationships between parts of documents of the first kind and associated parts of documents of the second kind, the first phase producing a modified translation file; and (c) in a second phase, using the modified translation file to convert the input document into the output document. Step (c) can include: (c1) activating the mapping functionality; and (c2) using the mapping functionality to provide references in the output document that associate parts of the output document with parts of the input document.

Related apparatus and computer readable media are also described herein.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 shows a known technique for transforming an XML document into another document, such as an HTML document.

Fig. 2 shows an exemplary data processing application that includes mapping between structured data and a visual surface.

Fig. 3 shows an exemplary solution file used in conjunction with a solution module shown in Fig. 2.

Fig. 4 shows an exemplary mapping module used in the data processing application of Fig. 2.

Fig. 5 shows an example of the annotation of an XSLT excerpt with mapping functions, and the subsequent execution of those mapping functions.

Fig. 6 illustrates an exemplary mapping between nodes of a visual surface and nodes of associated structured data.

Fig. 7 shows an entity relationship diagram that illustrates phases 1 and 2 of an annotation procedure used in the mapping module of Fig. 4.

Fig. 8 shows an entity relationship diagram that illustrates coupling between structured data and an associated visual surface, and the use of that coupling to enable editing operations.

Fig. 9 shows an exemplary apparatus for implementing the data processing application shown in Fig. 2.

Fig. 10 shows an exemplary user interface (UI) for designing an electronic form.

Fig. 11 shows an exemplary user interface (UI) for editing the electronic form created in the user interface of Fig. 10.

Fig. 12 shows an exemplary procedure for generating annotations in an electronic form that provide mapping back to underlying data, and for subsequently editing the electronic form having those annotations.

Fig. 13 shows an exemplary computing environment for implementing the data processing application shown in Fig. 2.

The same numbers are used throughout the disclosure and figures to reference like components and features. Series 100 numbers refer to features originally found in Fig. 1, series 200 numbers refer to features originally found in Fig. 2, series 300 numbers refer to features originally found in Fig. 3, and so on.

DETAILED DESCRIPTION

This disclosure pertains to the rendering and editing of information based on structured input data. To provide a concrete framework for discussion, this disclosure will specifically describe the transformation of hierarchically organized data expressed in a markup language into an electronic form that can be visually rendered and edited by an end user. Exemplary electronic forms can include a timesheet, work order, travel log, and so on. However, the concepts described herein also have application to other data processing applications besides electronic forms processing.

This disclosure is organized as follows. Section A of this disclosure describes an exemplary design strategy used to provide mapping between structured data and a visual surface. Section B describes an exemplary implementation of the design strategy discussed in Section A. Section C describes an exemplary method of operation of the implementation described in Section B. And Section D describes an exemplary computing environment that can be used to provide the implementation described in Section B.

A. Exemplary Design Strategy

Overview of Design Strategy

Fig. 2 shows an overview of a data processing application 200 for rendering structured documents. This data processing application 200 is exemplary. The mapping

mechanism described herein can be implemented in many different kinds of systems and environments besides the data processing application 200 shown in Fig. 2.

By way of overview, the data processing application 200 processes structured data 202 expressed in a markup language, transforms this structured data 202 using a solution module 204 to produce transformed information, and presents a rendering of a visual surface 206 on an output device based on the transformed information. An editing user 208 interacts with the visual surface 206, as indicated by arrow 210, using, for instance keyboard 212, mouse device 214, or some other input device. The visual surface 206 can constitute the presentation of an electronic form having data entry fields associated with the structured data 202. In this case, the editing user 208's interaction 210 can involve the editing user 208 filling information into the data entry fields of the electronic form, such as by entering information into various text boxes, check boxes, etc.

Each of the above-described principal features – structured data 202, solution module 204, and visual surface 206 – will be described in greater detail below.

To begin with, the structured data 202 can be represented in the Extensible Markup Language (XML). XML is a subset of the Standard Generalized Markup Language (SGML) that enables developers to create customized tags that describe the meaning of data, as opposed to the presentation of data. An XML document is composed of XML elements, each of which includes a start tag (such as <author>), an end tag (such as </author>), and information between the two tags (which is referred to as the content of the elements). An element may include a name-value pair (referred to as an attribute) related by an equal sign that modifies certain features of the element (such as MONTH = "May"). The elements in an XML document have a hierarchical relationship to each other that can be represented as a data tree 216. The elements in the data tree 216 are also commonly referred to as "nodes." A so-called XML schema (not illustrated in Fig.

2) provides a formal specification that defines the types of elements and the organization of elements that should appear in an XML document in order for that document to be considered so-called well formed.

The solution module 204 includes a transformation module 218. The purpose of the transformation module 218 is to transform the structured data 202 into the visual surface 206. The transformation module 218 can perform this task using so-called style sheets, such as style sheets provided by Extensible Stylesheet Language Transformation (XSLT). XSLT transforms the structured data 202 into a format appropriate for presentation, such as the Hypertext Markup Language (HTML), Extensible Hypertext Markup Language (XHTML), Dynamic HTML (DHTML), etc. In other words, documents expressed in XML include tags that are particularly tailored to convey the meaning of the data in the documents. The XSLT conversion converts the XML documents into another markup language in which the tags pertain to the visual presentation of the information contained in the documents. (To facilitate discussion, the following description assumes the use of HTML to render the documents; however, other presentation-oriented markup languages can be used to render the documents.) Because HTML is a markup language, it can be conceptualized as a view tree 220 that includes a hierarchical organization of nodes, as in the case of data tree 216. The reader is referred to the World Wide Web Consortium's specifications for background information regarding XML and XSLT.

A mapping module 222 enables nodes in the view tree 220 to be mapped to corresponding nodes in the data tree 216. Further, the mapping module 222 enables nodes in the data tree 216 to be mapped to corresponding nodes in the view tree 220. The mapping of nodes in the view tree 220 to nodes in the data tree 216 allows the solution module 204 to correlate editing operations performed on the visual surface 206 with

corresponding nodes in the underling structured data 202. This allows the solution module 204 to store information entered by the editing user 208 at appropriate locations within the structured data 202 during an editing session. Fig. 2 represents the above-described two-way mapping using arrows 224 and 226. More specifically, arrow 224 represents the mapping of information in the view tree 220 back to associated information in the data tree 216. Arrow 226 represents mapping of information in the data tree 216 to information in the view tree 220. The present discussion particularly addresses the mapping represented by arrow 224.

By way of broad overview, the mapping module 222 provides mapping between the visual surface 206 and the structured data 202 by adding annotations to the view tree 220 used to render the visual surface 206. These annotations serve as references which point back to specific locations in the data tree 216. Fig. 2 represents the annotation of the visual surface 206 by showing an annotated HTML document 228 being output from the solution module 204.

The visual surface 206 itself has an appearance that is determined by both the information contained in the structured data 202 as well as the effects of the XSLT transformation provided by the transformation functionality 218. Generally, in the case of electronic forms, the visual surface 206 typically includes a hierarchical structure which is related to the hierarchical structure in the structured data 202. For instance, an exemplary electronic form 230 includes multiple sections pertaining to different topics that reflect the topics in the structured data 202. (However, it is not necessary to have a one-to-one direct correspondence between the organization of the structured data 202 and the organization of the visual surface 206; in other words, the transformation of the structured data 202 to the visual surface 206 is generally considered non-isomorphic). Each section in the exemplary electronic form 230 can include one or more data entry

fields for received input from the editing user 208, such as data entry field 232. The data entry fields are also referred to herein as "editing controls." Different graphical components can be used to implement the editing controls, including text boxes, drop-down list boxes, lists boxes, option buttons (also referred to as radio buttons), check boxes, and so on. Figs 10 and 11, to be described in turn, provide an example of the visual appearance of an electronic form as it is being designed and edited, respectively.

Path 234 generally represents the routing of information entered via the electronic form 230 back to the structured data 202. In another words, the data entry fields in the electronic form 230 (such as data entry field 232) are associated with respective nodes in the data tree 216. Entry of information via electronic form 230 will therefore prompt the solution module 204 to route such information to appropriate storage locations in the data tree 216. Again, the linking between the electronic form 230 and the structured data 202 is provided by the mapping module 222.

The functionality provided by the solution module 204 is defined, in part, by a solution file, such as exemplary solution file 236 stored in storage 238. The solution file 236 essentially constitutes an electronic form template, providing all of the semantic information required to transform the structured data 202 into the visual surface 206. Different XML documents may have been created by, or otherwise refer to, different electronic form templates. Accordingly, different XML documents may have different solution files associated therewith. Various techniques can be used to retrieve a solution file that is associated with a particular XML document. For instance, an appropriate solution file can be retrieved based on URN (Uniform Resource Name) or URL (Uniform Resource Locator) information contained in the header of an input XML document. That header information links the input document to a corresponding solution file. A storage

240 represents an archive for storing one or more XML documents created by, or otherwise associated with, respective solution files.

Fig. 3 shows an exemplary composition of the solution file 236. As shown there, the solution file 236 contains a collection of files (302, 304, 306, 308, and 310) that together provide semantic information used, in part, to implement the solution module 204. This collection of files can be packaged together. In one exemplary implementation, this collection of files is referred to using an extension .xsn. A form definition file 302, also called a manifest file, forms the centerpiece of the collection. The form definition file 302 contains information about all of the other files in the solution module 204. This file 302 is assigned the exemplary extension .xsf. A schema file 304 is used to constrain and validate the structured data 202. This file is assigned the exemplary extension .xsd. View files 306 provide presentation logic files that are used to present, view, and transform the structured data 202. These files therefore implement the transformation module 218 discussed in connection with Fig. 2. The view files 306 can include multiple files corresponding to multiple possible views (i.e., visual surfaces 206) that the editing user 208 can select from. The view files 306 are assigned the exemplary extension .xsl. A default data file 308 contains default data that can be displayed in a view for fields that have not been explicitly defined by the editing user 208. This file 308 is assigned the exemplary extension .xml. Finally, business logic files 310 provide programming code used to implement specific editing behavior, data validation, event handlers, control of data flow, and other features. Such programs can be written in any kind of language, such as the scripting language provided by JScript® or VBSCRIPT scripting language. In this case, these files are assigned the exemplary extensions .js or .vb (for JScript® and VBSCRIPT scripting languages, respectively).

The Mapping Module

Fig. 4 shows an exemplary composition of the mapping module 222 introduced in Fig. 2. The mapping module 222 receives the structured data 202 and adds annotations to it to produce the annotated HTML document 228 (or other kind of annotated transformed document). The mapping module 222 performs this task in two phases: phase 1 and phase 2. In phase 1, the mapping module 222 takes arbitrary XSLT information (or other kind of transformation instructions) and adds mapping functions to it. These mapping functions are inserted at particular locations within the XSLT information. These mapping functions provide functionality that, when activated, generate references to specific locations within the structured data 202. However, in the first phase itself, the mapping module 222 simply inserts these mapping functions within the XSLT information; that is, in this phase, the mapping module 222 does not execute the functions to return the actual references that point to appropriate parts of the structured data 202. In the second phase, the mapping module 222 executes the mapping functions to provide actual values for the references.

The first phase is performed on the XSLT information itself, outside the context of the processing of any specific XML document. More specifically, the first phase can be performed once, for instance, after an electronic form has been newly created or modified. This has the effect of modifying the XSLT information associated with the newly created or modified electronic form by adding mapping functions to it. The second phase, by contrast, is performed each time a particular XML document is rendered. In the second phase, the mapping functions within the XSLT information are executed with respect to a particular XML document, to thereby produce an output HTML document (or other kind of output document) that has references inserted throughout it that point back to various locations in the particular XML document. Thus,

to summarize, the first phase is performed once upon the creation or modification of the XSLT information, whereas the second phase is performed each time a particular XML document is rendered. The second phase can also be referred to as the "runtime" phase, as it is performed when a particular XML document is rendered. Additional aspects of the above-described functionality will be described with reference to the logic illustrated in Fig. 4.

To begin with, the first phase acts on so-called arbitrary XSLT information 402. The XSLT information 402 is arbitrary in the sense that it is not prepared specifically with the annotation mechanism described above in mind; in other words, the XSLT information 402 can constitute any kind of XSLT information produced by any process in any environment. The arbitrary XSLT information 402 can serve a conventional role of converting an XML document 404 into an HTML document 406 (or other kind of the document). The resultant HTML document 406 would not contain any pointer annotations, and hence would not have the capability of mapping a resultant visual surface back to the originating XML document 404.

Phase 1 of the mapping module 222 takes this arbitrary XSLT information 402 and adds mapping functions to it. An annotation module 408 performs this role. The output of the annotation module 408 represents annotated XSLT information 410 having the mapping functions added thereto. The annotated XSLT information 410 can be stored in a storage (for example, a cache storage 412) for later use in phase 2 (the runtime portion of the procedure).

In one implementation, the mapping functions added by the annotation module 408 can be implemented as so-called XSLT extension functions. More specifically, XSLT provides a collection of tools to accomplish certain tasks. However, the range of functions that can be performed with unsupplemented XSLT is limited; XSLT cannot

perform some tasks very well, and cannot perform other tasks at all. Extension functions constitute references within the XSLT information that act as triggers to call some extended functionality to execute tasks not provided within XSLT itself. In the instant case, the extension functions perform the task of adding references to the XSLT information that point back to respective locations in the structured data 202. To repeat, however, these mapping functions are not executed in phase 1; rather, in phase 1, they are merely inserted in the XSLT information 402 at appropriate locations.

Different strategies can be used to govern where to insert the mapping functions within the XSLT information 402. These strategies may differ from one processing environment to the next, because different processing environments may involve the processing of different types of documents having different characteristics. In the present case, an electronic form often has a nested structure. For instance, a section of the electronic form may contain a subsection, and that subsection may have its own respective subsection(s). Any of these sections and subsections can have data entry fields included therein. For example, an electronic form can include a table that defines a primary section. That table, in turn, can include plural subsections (e.g., rows), and each row can contain plural data entry fields. In this context, a so-called outer mapping can be used to identify a certain section or subsection in the electronic form. A so-called inner mapping can be used to specifically identify a data entry field within that section or subsection. The inner mappings thus provide the specific bindings between the data entry fields in the electronic form and the respective nodes of the structured data 202 associated with the data entry fields. The outer mappings provide information regarding the scope (e.g., extent) of a section or subsection that may include one or more inner mapping data entry points. In the context of the above example pertaining to the rendering of a table in the electronic form, outer mappings can be used to demarcate the table itself, as well as

individual rows within the table. Inner mappings can be used to identify data entry fields within the table.

Still more specifically, the annotation module 408 can add outer mappings in the XSLT information 402 at locations representative of context changes. There are two ways to change context in XSLT: (1) using an "apply-templates" instruction; and (2) using a "for-each" instruction. The "apply-template" instruction causes the output flow of the XSLT processing to move to a new template, which is evaluated in the new context. To mark these context changes, the annotation module 408 annotates all direct children of template nodes with mapping function calls requesting the respective IDs of the current context. For the "for-each" instruction, the annotation module 408 causes the output flow to move to the child of the "for-each" node. In this case, the annotation module 408 annotates all direct children of the "for-each" nodes with mapping function calls to the respective IDs of the current context. Generally, as is well known, the "apply-template" instruction applies a template rule deemed most suitable for processing a current node and its children. The "for each" instruction performs specified actions for a collection of nodes that satisfy a selection expression.

The annotation module 408 can add inner mappings in those cases where XSLT pulls the contents of XML nodes of the data tree 216 directly into the view tree 220. This content can be mapped directly from the view tree 220 back to the XML nodes in the data tree 216 from which they were pulled. More specifically, XSLT pulls out content using the "value-of" and "copy-of" instructions used in XSLT. The annotation module 408 marks these content grabs by adding mapping function calls requesting the IDs of the respective XML nodes in the data tree 216 being referenced. Annotations are not generated if the mapping is ambiguous. This could happen if the "value-of" instruction refers to more than one XML node in the data tree 216. Generally, as is well known, the

"copy-of" instruction of XSLT copies all aspects (attributes, tags, children, etc.) of identified nodes into a result tree. The "value-of" instruction in XSLT converts the identified nodes to a string and adds this string to the result tree.

The annotation module 408 automatically adds the outer and inner mappings based on the above-described guidelines (that is, by adding mapping functions where the above-described XSLT instructions occur). This automatic annotation may not be sufficient for all situations. To address these cases, XSLT authors can "manually" modify the XSLT to include mapping functions at locations selected by the XSLT authors.

Phase 2 of the mapping procedure involves executing the mapping functions added in phase 1 to return specific references to nodes in the data tree 216. A runtime XSLT module 414 performs this function to yield instantiated annotated XSLT information 416 having specific references added thereto. The ultimate output of the runtime XSLT module 414 is the annotated HTML document 228 (or a document expressed in some other structured format). More specifically, the extension functions added in phase 1 provide XPath references to namespaced functions. When the XSLT information 402 is processed at runtime, the runtime XSLT module 414 reads the namespaced functions and calls them, passing a node list as a parameter. The runtime XSLT module 414 analyzes this node list, ensures that it is unambiguous (e.g., that it contains only one node), and returns identifiers for these nodes. The runtime XSLT module 414 writes these identifiers to a result tree, thus building the HTML document 228 having mapping references added thereto.

Fig. 5 provides an example of the operation of the mapping module 222 shown in Fig. 4. Excerpt 502 represents an original excerpt of XSLT information, corresponding to the arbitrary XSLT information 402 shown in Fig. 4. Excerpt 504 represents the

original XSLT information 402 with mapping functions added thereto, thus forming the annotated XSLT information 410 shown in Fig. 4. And excerpt 506 represents the XSLT information 402 having the mapping functions executed at runtime, thus forming the instantiated annotated XSLT 416 shown in Fig. 4. Functions 508 and 510 represent inner and output mapping functions, respectively, added to the XSLT information 402 in phase 1. Annotations 512 and 514 represent inner and output mapping references, respectively, added to the XSLT information 402 in phase 2 in response to the execution of the mapping functions in excerpt 504.

Fig. 6 shows a high-level exemplary depiction of the mapping between the nodes of the view tree 220 and the nodes of the data tree 216. For instance, the view tree 220 contains exemplary ID references 602, 604, 606, and 608 added to respective nodes. For instance, exemplary ID reference 602 is associated with node 610. This ID reference 602 points back to a node 612 in the data tree 216. In other words, this reference 602 indicates that the node 612 in the data tree 216 contributed to the formation of node 610 in the view tree 220 through the transformative effects of the XSLT applied to the data tree 216. Node 610 in the view tree 220 may be associated with a data entry field in an electronic form. If this is the case, then knowledge of the linking between node 610 in the view tree 220 and node 612 in the data tree 216 allows the solution module 204 to route data entered into the electronic form via this data entry field to an appropriate location in the structured input data 202 for storage thereat.

Fig. 7 shows an entity relationship diagram 700 that illustrates the conceptual relationship between the generic mapping provided by phase 1 and the instantiated mapping provided by phase 2. (In this diagram, the "crow's feet" connector notation represents a one-to-many and many-to-one type of relationship between entities, as per convention.) That is, entity 702 represents the mapping functions added to the XSLT

information in phase 1 of the mapping procedure. The mapping functions in this phase provide general rules for applying mapping functions to specified types of XML nodes in a general group of possible XML documents. However, within this group, individual XML documents can vary in different ways. Hence, conceptually, the mapping functions provided in phase 1 apply to any XML document on a relatively general or abstract level. Entity 704, on the other hand, represents the application of the annotated XSLT to a specific XML document. This happens in phase 2 of the mapping procedure. In phase 2, the mapping functions are executed to return specific pointers in the context of the processing of a specific XML document, to ultimately generate an annotated HTML document. Accordingly, entity 704 represents a particular instance of the general range of possibilities represented by entity 702.

Structural Editing Using Mapping

Fig. 8 shows an entity relationship diagram 800 that illustrates the coupling between the data and visual aspects of the data processing application 200. This diagram also illustrates structural editing functionality used to edit the structured data 202. That is, the structural editing provides a mechanism that allows input received through the visual presentation of the electronic form to produce corresponding changes in the structured data 202. The structural editing therefore employs the above-described mapping functionality as an integral part thereof.

To begin with, the left-hand side of the entity relationship diagram 800 of Fig. 8 pertains to data handling aspects of the data processing application 200, and is referred to herein simply as data-side 802. The right-hand side 804 pertains to view handling aspects of the data processing application 200 associated with the visual surface 206, and is referred to herein simply as view-side 804. By way of overview, the view-side 804 shows functionality for selecting a particular part of the visual surface 206. In the context

of an electronic forms application, this may represent the selection of a particular field in the electronic form by the editing user 208. Changes to the selected field of the electronic form may prompt the data processing application 200 to make corresponding changes in the structured data 202 which is mapped to the selected field. The data-side 802 shows functionality for identifying the particular nodes in the structured data 202 (e.g., XML data) that are mapped to the selected field. In summary, the view-side 804 selects a part of the visual surface 206 (e.g., expressed in HTML) and the data-side 802 finds the XML nodes corresponding to the selected part of the visual surface 206.

More specifically, ViewNode entity 806 represents a node in the visual surface 206 (e.g., a node in the view tree 220), and XmlDataNode entity 808 represents a node in the structured data 202 (e.g., a node in the data tree 216). The loop at the top of ViewNode entity 806 represents that a collection of nodes in the view-side 804 forms a hierarchical tree (e.g., the view tree 220). The loop at the top of the XmlDataNode 808 entity likewise means that a collection nodes in the data-side 802 forms another hierarchical tree (e.g., the data tree 216). A horizontal line 810 that couples the ViewNode entity 806 to the XmlDataNode entity 808 indicates that the view tree 220 is mapped to the data tree 216. This same concept is conveyed by arrows 224 and 226 shown in Fig. 2.

The functionality for selecting a part of the visual surface 206 includes ViewRange entity 812 and ViewPointer entity 814. The ViewRange entity 812 refers to a tool used to select and identify content in the visual surface 206 to be edited. The ViewPointer entity 814 defines endpoints in a range associated with the ViewRange entity 812. A View entity 816 conceptually represents an aggregation of all aspects of the view-side 804; that is, the View entity 816 defines the visual surface 306 as an aggregation of all of the entities shown in the view-side 804.

The entities in the data-side 802 identify the XML nodes that are mapped to the part of the visual surface 206 selected by the ViewRange entity 812 and the ViewPointer entity 814. Generally, the XML editing mechanism operates by: (1) determining a part of the view-side 804 selected by the editing user 208 (defining a "selected part"); (2) determining an associated part of the structured data 202 in the data-side 802 that is linked to the selected part in the view-side 804 (defining an "associated part"); (3) determining whether any editing rules apply to the associated part in the data-side 802 by matching pattern information in the associated part to rules contained in the forms definition file 302 of Fig. 3; and (4) if there are rules that pertain to the associated part, applying those rules to the editing operation involving the selected part of the view-side 804. Generally, the rules can specify the behavior of the editing operation, such as what parts of the view 816 and associated structured data are selectable, editable, etc. The rules can also specify whether certain behavioral features apply to the editing operation, such as proofing, spelling correction, auto-completion, data validation, and so on. Since the behavior of the view-side 804 is determined by performing matching of patterns within the structured data 202, this technique of editing can be viewed as "data-side matching."

A more detailed explanation of the above-described data-side matching is provided as follows. By way of introduction, an electronic form presented in the view-side 804 includes a collection of "editing controls," or simply "controls." These editing controls allow the editing user 208 to enter information into the electronic form using different techniques depending on the nature of the editing controls. Exemplary editing controls include text boxes, rich text boxes, etc. So-called "editing components" represent functionality associated with respective editing controls. The editing components specify how the structured data 202 can be edited in response to the editing

user 208's manipulation of the editing controls. In one exemplary implementation, the creation of an editing control also prompts the creation of a corresponding editing component that specifies how this editing control interacts with the structured data. For example, an xCollection editing component is associated with repeating section and repeating table editing controls, an XOptional editing component is associated with an optional section editing control, an xTextList editing component is associated with a plain list, bulleted list, or numbered list editing controls, an xField editing component is associated with a rich text box and text box editing controls, and an xImage editing component is associated with a picture editing control.

An xsf:EditWith entity 818 specifies the behavior of an editing control as determined by its editing component. More specifically, this entity 818 specifies that the editing control uses a given editing component and it provides the corresponding parameters to the editing component to determine its exact behavior. An xsf:XmlToEdit entity 820 defines the location of the editing control within the view-side 804, as determined by XML mapping. In brief, the xsf:EditWith entity 818 defines "what to do," and the "xsf:XmlToEdit" entity 820 defines "where to do it."

Consider, for example, the following XML document.

```
<root>
   <issues>
      <issue author = "Pete">
         <title> HTML versa XML tables </title>
         </description>
            <textItem>some text goes here</textItem>
            <textItem>more text goes here</textItem>
         </description>
         <workItems>
            <workItem description = "create visuals" effort = "2"/>
            <workItem description = "create visuals" effort = "2"/>
         </workItems>
         <notes>type here</notes>
```

```
            </issue>
        </issues>
    </root>
```

The following exemplary editing functionality provided in the forms definition file 302 employs the xTextList editing component to edit "textItem" XML nodes associated with the above-identified XML document.

```
<xsf:xmlToEdit item = "description/textItem">
    <xsf:editWith component = "xTextList" type = "formatted"/>
<xsf:xmlToEdit>
```

The following exemplary editing functionality provided in the forms definition file 402 employs the xCollection editing component to edit "workItem" XML nodes associated with the above-identified XML document.

```
<xsf:xmlToEdit anme = "workItem" Item = "workItems/workItem" container = "workItems">
  <xsf:editWith component = "xCollection">
      <xsf:fragmentToInsert>
          <xsf:chooseFragment>
              <workItem description = "create visuals" effort = "2"></workItem>
          </xsf:chooseFragment>
      <x/sf:fragmentToInsert>
  </xsf:editWith>
<xsf:/xmlToEdit>
```

As noted in the above examples, the editing component xCollection has attributes "container" and "item" associated therewith, whereas the editing component xTextList has only attribute "item" associated therewith. The two lines emanating from the top of the xsf:XmlToEdit entity 820 represents the item and collection attributes. The container attribute corresponds to an XPath match pattern which determines the context in which

the editing control will be selectable and its actions enabled. If the current context (e.g., view selection or insertion point in the view-side 804 is within some HTML element which maps back to an XML node which satisfies this container match pattern, then the editing control is enabled. More specifically, an exemplary procedure for finding the container XML node is as follows: (i) Start from the current selection within the visual surface 206; and (ii) Continue up the HTML ancestors, and seek an HTML node that maps to an XML node satisfying the container XPath match pattern. If found, a container HTML node and corresponding container XML node are thereby provided. In one exemplary implementation, it does not suffice for the container XML node to exist. Actions are enabled only when the current selection in the view-side 804 is within an HTML element which maps to the container XML node. The item attribute is also an XPath match pattern. It specifies the XML nodes to be edited using editing components indicated in the contained <editWith> elements. The item attribute is used to identify an XML node in a manner similar to that stated above with respect to the container attribute.

The fragmentToInsert parameter in the above-identified example of the xCollection editing component specifies the XML fragment which is to be inserted into the source XML. This parameter is set using a fragmentToInsert element, which is a child of the editWith element. Further, the fragmentToInsert element contains one or more chooseFragment child elements, which specify a choice between different versions of the fragment. The fragment itself is specified inline, within the chooseFragment elements.

Finally, an XPath entity 822 and XPathMatch entity 824 represent the above-described use of XPath functionality and pattern matching to identify the XmlDataNode entity 808 associated with the part of the visual surface selected in the view-side 804.

B. Exemplary Apparatus for Implementing Mapping

Fig. 9 shows an overview of an exemplary apparatus 900 for implementing the data processing application 200 shown in Fig. 1. The apparatus 900 includes a computer 902 that contains one or more processing units 904 and memory 906. Among other information, the memory 906 can store an operating system 908 and the above-described data processing application 200, identified in Fig. 9 as a forms application 910. The forms application 910 can include data files 912 for storing the structured XML data 202, and solution module 914. As noted above, a solution module 914 comprises logic that specifies the appearance and behavior of the visual surface 206 (as was described in connection with Fig. 2). The logic provided by solution module 914 is, in turn, determined by a solution file (such as a solution file 236 composed of the files shown in Fig. 5). The computer 902 is coupled to a collection of input devices 916, including the keyboard 212, mouse device 214, as well as other input devices 918. The computer 902 is also coupled to a display device 920.

In one exemplary implementation, the forms application 910 includes a design mode and an editing mode. The design mode presents design UI 922 on the display device 920 for interaction with a designing user 924. The editing mode presents editing UI 926 on the display device 920 for interaction with the editing user 208. In the design mode, the forms application 910 creates an electronic form 928, or modifies the structure of the electronic form 928 in a way that affects its basic schema. In other words, the design operation produces the solution file 236 that furnishes the electronic form 928. In the editing mode, the editing user 208 uses the electronic form 928 for its intended purpose – that is, by entering information into the electronic form 928 for a business-related purpose or other purpose.

In the design mode, the forms application 910 can be configured to depict the electronic form 928 under development using a split-screen display technique. More specifically, a forms view portion 930 of the design UI 922 is devoted to a depiction of the normal appearance of the electronic form 928. A data source view portion 932 of the visual surface is devoted to displaying a hierarchical tree 934 that conveys the organization of data fields in the electronic form 928.

Fig. 10 shows an exemplary design UI 922 that illustrates the allocation of the visual surface 206 into the forms view portion 930 and the data source view portion 932. As described above, the forms view portion 930 contains a depiction of the normal appearance of the form 928 – in this case, exemplary form 1002. The exemplary electronic form 1002 shown in Fig. 9 includes a plurality text box entry fields (e.g., fields 1004, 1006, 1008, 1010, 1012, and 1014. The data source view portion 932 includes the hierarchical tree 934 showing the nested layout of the text fields (1004-1014) presented in the form 1002.

The forms application 910 provides multiple techniques for creating the electronic form 1002. According to one technique, the electronic form 1002 can be created from scratch by building the electronic form 1002 from successively selected editing controls. The exemplary electronic form 1002 shown in Fig. 10 is entirely constructed using the text entry boxes (1004-1014), but other electronic forms can include other kinds of entry fields (i.e., editing controls), such as drop-down list boxes, list boxes, option button, check boxes, and so on. In another technique, the electronic form 1002 can be created based on any pre-existing .xsd schema document loaded into the forms application 910. The .xsd schema is an XML file that defines the structure and content type of the XML files that are associated with it. In another technique, the electronic form 1002 can be created based on an XML document. The forms application 910 will then create a

schema based on the information in the input XML file. In another technique, the electronic form 1002 can be created based on a database schema. In this case, the forms application 910 will extract the schema of the data and convert that record set to XML representation. Still other techniques can be used to create electronic forms.

Once a form has been created, its design (and associated schema) can be further modified. For example, the forms application 910 allows the designing user 924 to modify existing editing controls used in the electronic form 1002, or add additional editing controls. For instance, the UI panel 1016 allows the designing user 924 to modify the editing control associated with the company data field 1006. Selecting a particular control type – for example a check box – will prompt the forms application 910 to substitute the control type previously used to receive company related information with the newly selected control type. There are many other ways to modify the electronic form 1002.

The creation of the form 1002 also creates an associated solution file. The solution file effectively forms a template that can be archived and subsequently used in a business (or other environment). Fig. 11 demonstrates an exemplary use of the form 1002. More specifically, this figure shows the presentation of the electronic form 1002 in the editing mode of operation of the forms application 910. In this case, the editing user 208 is filing text into the text entry fields in the UI presentation 926. For instance, the editing user 208 is currently entering text 1102 into the text field 1010. The editing user 208 can select a particular part of the electronic form 1002 in a conventional manner, such as by pointing to and clicking on a particular field in the electronic form 1002 using the mouse device 214.

As described in Section A of this disclosure, data entry fields (1004-1014) in the electronic form 1002 are mapped to underlying structured data 202 – in this case, XML

document 1104. This mapping is achieved via annotations added to the HTML document used to render the electronic form 1002. More specifically, the annotations act as references which point to particular parts of the XML document 1104 associated with the data entry fields (1004-1014) in the electronic form 1002. Through this mechanism, the data entered by the editing user 208 is routed back to the XML document 1104 and stored in its data structure at appropriate locations. This mapping functionality is represented in Fig. 11 by the arrow 1106.

C. Exemplary Method of Operation

Fig. 12 shows an exemplary procedure 1200 for creating and editing an electronic form. The procedure 1200 can be implemented in software or firmware, or a combination of software and firmware.

Phase 1 of the procedure 1200 includes steps 1202, 1204, and 1206. Step 1202 involves receiving XSLT information. This step 1202 might correspond to receiving an XSLT file created in response to the creation or modification of an electronic form, or from some other source. The XSLT information is arbitrary in the sense that it does not need to be developed specifically to accommodate the annotation functionality which is subsequently applied to it. An exemplary technique for creating an XSLT file in the context of electronic forms processing is described in commonly assigned U.S. Patent Application No. 10/395,506, filed on March 24, 2003, entitled "System and Method for Designing Electronic Forms," naming Christina Fortini, Jean D. Paoli, Laurent Mollicone, Bulusu Krishna Mohan, and Alessandro Catorcini, which is incorporated herein by reference in its entirety. Step 1204 involves automatically annotating the arbitrary XSLT by adding mapping functions to it. As described above, these mapping functions can constitute extension functions added to the XSLT information at inner and

out mapping locations. Step 1206 involves caching the annotated XSLT for later retrieval and use. The XSLT author can also manually add mapping functions to the XSLT information to supplement the automatic annotations added to the XSLT information.

Phase 2 of the procedure 1200 involves steps 1208, 1210, and 1212. Step 1208 entails receiving an XML document to be processed using the annotated XSLT information. The XML document can be considered arbitrary, like the XSLT information, in the sense that it does not have to be structured to accommodate the annotation procedure that is subsequently applied to it; any XML document will suffice. Step 1210 entails executing the mapping functions in the annotated XSLT information to return specific reference values that point back to the structured data 202. Step 1212 entails outputting an annotated HTML document (or some other markup language document) for display. The HTML document is annotated by including references that point back to respective locations within the structured input data 202.

Following display of the annotated HTML document, the editing user 208 can edit the displayed electronic form. Steps 1214, 1216, and 1218 pertain to this editing operation. In step 1214, the forms application 910 receives the editing user 208's commands to execute an editing operation. These commands may be the result of the user pointing to a particular part of the visual surface 206 using the mouse device 214 and then inputting data into data entry fields using the keyboard 212. Other ways of editing the electronic form can be used. Step 1216 involves routing the editing user 208's input back to the source XML document for storage at appropriate locations in the structured XML data. To perform this routing, the above-described mapping annotations are used to link selected parts of the visual surface with associated parts of the XML source data. Finally, in step 1218, the procedure 1200 involves updating the visual surface 206 to

reflect the user's editing operations with respect to the visual surface 206. An exemplary technique for performing step 1218 is described in commonly assigned Application No. 10/404,312, filed on March 31, 2003, entitled "System and Method for Incrementally Transforming and Rendering Hierarchical Data Files," naming Prakash Sikchi, Dragos Barac, Ranjan Aggarwal, and Steven J. Mooney as inventors, and incorporated herein by reference in its entirety.

D. Exemplary Computer Environment

Fig. 13 illustrates one example of a computing environment 1300 within which the above-described forms application 910 can be either fully or partially implemented. The computing environment 1300 includes the general purpose computer 902 and display device 920 discussed in the context of Fig. 9. However, the computing environment 1300 can include other kinds of computer and network architectures. For example, although not shown, the computer environment 1300 can include hand-held or laptop devices, set top boxes, programmable consumer electronics, mainframe computers, gaming consoles, etc. Further, Fig. 13 shows elements of the computer environment 1300 grouped together to facilitate discussion. However, the computing environment 1300 can employ a distributed processing configuration. In a distributed computing environment, computing resources can be physically dispersed throughout the environment.

Exemplary computer 902 includes one or more processors or processing units 904, a system memory 906, and a bus 1302. The bus 1302 connects various system components together. For instance, the bus 1302 connects the processor 904 to the system memory 906. The bus 1302 can be implemented using any kind of bus structure or combination of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a

variety of bus architectures. For example, such architectures can include an Industry Standard Architecture (ISA) bus, a Micro Channel Architecture (MCA) bus, an Enhanced ISA (EISA) bus, a Video Electronics Standards Association (VESA) local bus, and a Peripheral Component Interconnects (PCI) bus also known as a Mezzanine bus.

Computer 902 can also include a variety of computer readable media, including a variety of types of volatile and non-volatile media, each of which can be removable or non-removable. For example, system memory 906 includes computer readable media in the form of volatile memory, such as random access memory (RAM) 1304, and non-volatile memory, such as read only memory (ROM) 1306. ROM 1306 includes an input/output system (BIOS) 1308 that contains the basic routines that help to transfer information between elements within computer 902, such as during start-up. RAM 1304 typically contains data and/or program modules in a form that can be quickly accessed by processing unit 904.

Other kinds of computer storage media include a hard disk drive 1310 for reading from and writing to a non-removable, non-volatile magnetic media, a magnetic disk drive 1312 for reading from and writing to a removable, non-volatile magnetic disk 1314 (e.g., a "floppy disk"), and an optical disk drive 1316 for reading from and/or writing to a removable, non-volatile optical disk 1318 such as a CD-ROM, DVD-ROM, or other optical media. The hard disk drive 1310, magnetic disk drive 1312, and optical disk drive 1316 are each connected to the system bus 1302 by one or more data media interfaces 1320. Alternatively, the hard disk drive 1310, magnetic disk drive 1312, and optical disk drive 1316 can be connected to the system bus 1302 by a SCSI interface (not shown), or other coupling mechanism. Although not shown, the computer 902 can include other types of computer readable media, such as magnetic cassettes or other magnetic storage

devices, flash memory cards, CD-ROM, digital versatile disks (DVD) or other optical storage, electrically erasable programmable read-only memory (EEPROM), etc.

Generally, the above-identified computer readable media provide non-volatile storage of computer readable instructions, data structures, program modules, and other data for use by computer 902. For instance, the readable media can store the operating system 908, one or more application programs 1322 (such as the forms application 910), other program modules 1324, and program data 1326.

The computer environment 1300 can include a variety of input devices. For instance, the computer environment 1300 includes the keyboard 212 and a pointing device 214 (e.g., a "mouse") for entering commands and information into computer 902. The computer environment 1300 can include other input devices (not illustrated), such as a microphone, joystick, game pad, satellite dish, serial port, scanner, card reading devices, digital or video camera, etc. Input/output interfaces 1328 couple the input devices to the processing unit 904. More generally, input devices can be coupled to the computer 902 through any kind of interface and bus structures, such as a parallel port, serial port, game port, universal serial bus (USB) port, etc.

The computer environment 1300 also includes the display device 920. A video adapter 1330 couples the display device 920 to the bus 1302. In addition to the display device 920, the computer environment 1300 can include other output peripheral devices, such as speakers (not shown), a printer (not shown), etc.

Computer 902 can operate in a networked environment using logical connections to one or more remote computers, such as a remote computing device 1332. The remote computing device 1332 can comprise any kind of computer equipment, including a general purpose personal computer, portable computer, a server, a router, a network computer, a peer device or other common network node, etc. Remote computing device

1332 can include all of the features discussed above with respect to computer 902, or some subset thereof.

Any type of network can be used to couple the computer 902 with remote computing device 1332, such as a local area network (LAN) 1334, or a wide area network (WAN) 1336 (such as the Internet). When implemented in a LAN networking environment, the computer 902 connects to local network 1334 via a network interface or adapter 1338. When implemented in a WAN networking environment, the computer 902 can connect to the WAN 1336 via a modem 1340 or other connection strategy. The modem 1340 can be located internal or external to computer 902, and can be connected to the bus 1302 via serial I/O interfaces 1342 other appropriate coupling mechanism. Although not illustrated, the computing environment 1300 can provide wireless communication functionality for connecting computer 902 with remote computing device 1332 (e.g., via modulated radio signals, modulated infrared signals, etc.).

In a networked environment, the computer 902 can draw from program modules stored in a remote memory storage device 1344. Generally, the depiction of program modules as discrete blocks in Fig. 13 serves only to facilitate discussion; in actuality, the programs modules can be distributed over the computing environment 1300, and this distribution can change in a dynamic fashion as the modules are executed by the processing unit 904.

Wherever physically stored, one or more memory modules 906, 1314, 1318, 1344, etc. can be provided to store the forms application 910 programming code.

Although the invention has been described in language specific to structural features and/or methodological acts, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or acts described.

Rather, the specific features and acts are disclosed as exemplary forms of implementing the claimed invention.